Controlling User Interface Objects Through Pre- and Postconditions

Daniel F. Gieskens and James D. Foley

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-028

June 1991

Abstract

We have augmented user interface objects (i.e. windows, menus, buttons, sliders, etc.) with preconditions that determine their visibility and their enabled/disabled status and postconditions that are asserted when certain actions are performed on the object. Postconditions are associated with each functionally different action on the object. Attaching pre- and postconditions to interface objects provides several useful features, such as selective enabling of controls, rapid prototyping, and automatic generation of explanations and help text.

Introduction

Several techniques can be used to describe the dialogue of an application. Some of the best known techniques are transition diagrams, grammars and event languages. As discussed by Green [GREE86], the event model has a greater descriptive power than the former two. A particularly useful form of the event model is to associate pre- and postconditions with dialogue components (actions and/or interface objects). The preconditions of a dialogue component determine when the component would be enabled or activated, while the postconditions are used to describe changes in the state of the interface.

Pre- and postconditions were first used for user interface design by Green [GREE85] as part of a formal specification; They were not used at run-time to control the dialogue. In the User Interface Design Environment (UIDE) [FOLE89], pre- and postconditions are associated with application actions. They are used to describe partial semantics of application actions. These partial semantics are used for many purposes, including selective enabling of menu items, partial explanations of what an action does, providing context sensitive animated help [SUKA90], applying correctness-preserving transformations to the interface [FOLE87], checking the completeness and consistency of the interface, and dialogue sequencing.

By extending the UIDE mechanism to include all interface objects, we provide finer-grained control in UIDE, which used pre- and postconditions to control only the enabling of individual menu items. While the original UIDE used a set of predefined expressions in its conditions, we now allow arbitrary boolean predicates. The predicates can also be set by the application program, thereby affecting the state of the interface. Predicates can also have special variables which serve to communicate information between interface objects and the application.

Because pre- and postconditions contain semantic information about the dialogue components with which they are associated, they can be used to generate explanations about these dialogue components. For example, if a menu item has a precondition saying that there should be a selected object, a help tool can use this information to tell the user he has to select an object first. If a certain command is not available or if an interface object is disabled, pre- and postconditions of other commands and interface objects can be used to determine the sequence of actions needed to enable the command or the interface object [SENA89].

Pre- and postconditions not only describe semantics, but also encode dynamic behavior. A set of interface objects with pre- and postconditions can implement a complete dialogue without any additional program code.

Pre- and postconditions also form an interface between a dialogue component and its environment. Dialogue components become independent objects which communicate by means of pre- and postconditions. This is somewhat similar to the idea used in VUIMS [PITT90], where the interface consists of objects that communicate by sending tokens to each other. The main difference is that with the pre- and postconditions the messages are predicates that are posted on a ‘blackboard’, which makes the objects even more independent since they do not need to know who is interested in their information.

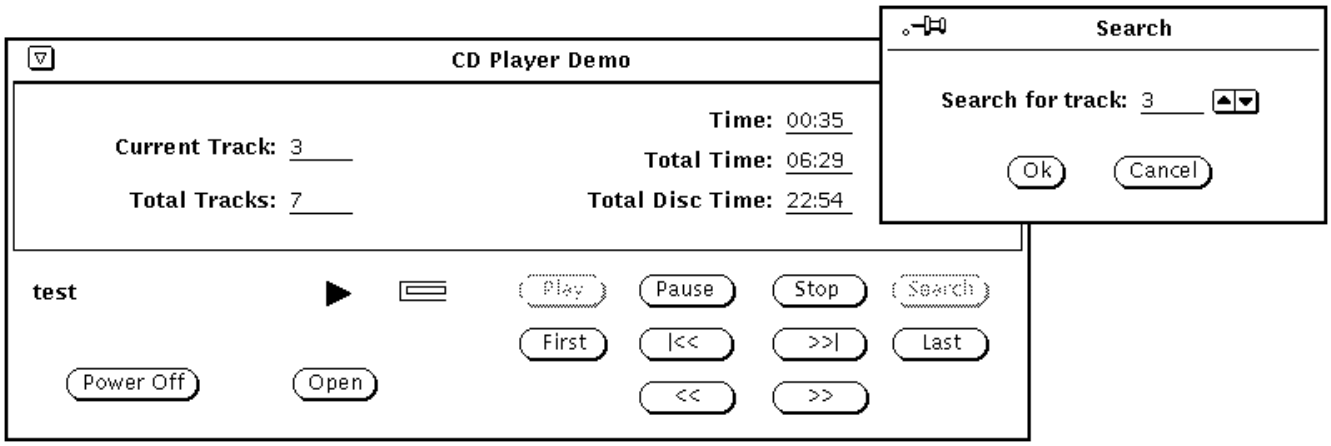

Figure 1 CD player interface

In this paper we explain how pre- and postconditions work, then we show how the system supporting this mechanism works. Finally we discuss how a user interface design tool, such as Sun Microsystems' Developers Guide [SUN90], can be enhanced with this mechanism to support a better prototyping environment.
In the rest of this article we will use the term widget when we are talking about interface objects. Several examples in this paper use a CD player application, which is a software interface used for playing music from a CD rom. This interface looks similar to a front panel of a real CD player (Figure 1).

Predicate Mechanism

In this system boolean predicates are posted on 'the Current State Blackboard' (CSB), which is a list of predicates that are currently true. The CSB is similar to a real-life blackboard: When a predicate becomes true it is written on the blackboard and when it becomes false it is simply erased. Predicates can be written to and removed from the CSB by means of postconditions as well as by the application itself. The application can also check if certain predicates are on the CSB. The CSB can be used to exchange semantic information between the application and the interface (UIMS) and thus facilitates separating the application and its interface.
Each predicate consists of a name and two arguments (we can capture any fact or relation in this form). Preconditions are boolean expressions and postconditions are lists of changes to be made in the CSB[1]. Each widget has two sets of preconditions and several postconditions, depending on the type of widget. One set of preconditions determines whether the widget is visible and the other determines whether it is enabled. A button, for instance, may be visible while not being enabled. On the other hand, a widget can never be enabled when it is not visible. Postconditions are associated with each functionally different action on the widget. Some widgets have only one possible functional action (like the select action on a button), while others have several possible actions.
Below are two examples, both taken from the CD player application. Example 1 shows how buttons can be greyed out automatically and example 2 shows how simple program dynamics can be encoded using pre- and postconditions. In the following examples we will describe widgets by stating their type and name (i.e. ***menu item*** *save_current_file*), followed by a list of label-value pairs that describe the widget's pre- and postconditions. The labels *pre visible* and *pre enable* stand for the visibility preconditions and the enable preconditions, respectively. The label *post <action>* (i.e. *post select*) describes the postconditions for that specific action on the widget.

Button *stop*
pre enable: *not status(*CD,STOPPED*)*
post select: *status(*CD,STOPPED*)*

Example 1 Selectively enabling of controls - stopping the CD is only useful when the CD is not already stopped.

Button *search*
post select: *popup(*SEARCH*)*

Popup window *search_track_dialogue*
pre visible: *popup(*SEARCH*)*

Example 2 Simple program dynamics - when the search button is selected the search-track dialogue window automatically pops up.

In the previous examples the predicates had literal (constant) arguments, which means the predicate and its arguments must be on the CSB in exactly the same form. Literal arguments are written in uppercase. Predicates can also have variable arguments, written in lowercase, in which case the

1. Postconditions consist of a list of predicates to be added to the CSB and a list of predicates to be removed from the CSB, an "add" list and a "delete" list respectively.

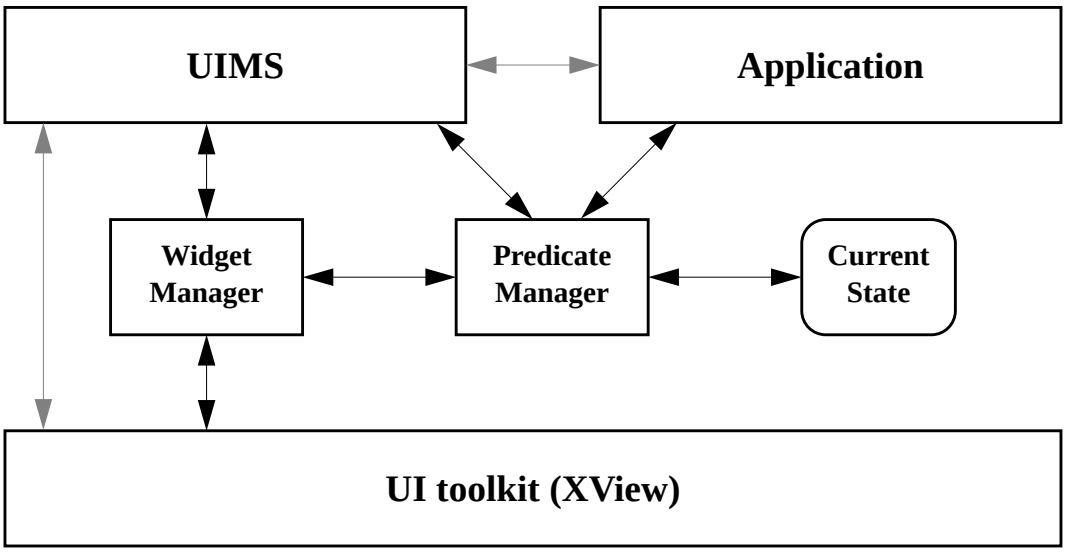

Figure 2 Architecture

argument matches with any text (i.e. *current_track(track)* matches with *current_track(*1*), current_track(*2*),* etc.). When a predicate containing one or more variables is matched against predicates in the CSB, both its truth-value and the literal(s) substituted for its variable(s) are returned. The literal values are retained, because the same variables can be used in the postconditions (as in Example 3).
In postconditions functions can be used to modify predicate arguments. Functions are used to perform simple computations, like increment and decrement (see Example 3), and to get information from the application.

Button *next_track*
pre enable: *current_track(curr) and total_tracks(total) and less_than(curr, total)*
post select: *current_track(inc(curr, 1))*

Example 3 Variables and functions - the variables *curr* and *total* get their values, based on what predicates are on the CSB. The value of *curr* is used in combination with the function *inc* to increment the current track.

Pre- and postconditions form an interface between a widget and its environment. Its preconditions determine when it is visible and enabled and the postconditions describe the changes that result from interactions with the widget. We defined special variables to increase the power of this form of communication between a widget and its environment. Special variables are variables in predicate arguments with a special meaning. The currently recognized special variables are:

- set_value: sets the value of the widget,
- get_value: gets the value from the widget,
- set_image: indicates the name of the bitmap that should be used by widgets that display images (icons, messages, etc.),
- upperbound: sets the upper limit of widgets with range capabilities (sliders, numeric text items, etc.), and
- lowerbound: sets the lower limit of widgets with range capabilities.

The special variables set_value, set_image, upperbound and lowerbound can only be used in preconditions, while get_value can only be used in postconditions. Especially useful are set_value and get_value, because these allow widgets to communicate their values (as in Example 4) with the rest of the interface and the application.

Numeric text item *search_track_item*
pre visible: *visible(*SEARCH_WINDOW*)*
pre enable: *current_track(set_value)*
post changed: *current_track(get_value)*

Example 4 Special variables - the variable *set_value* initializes the widget with the current track number. After the user has changed the value of the text item, the new value is used in the postconditions by means of *get_value.*

Normally, when a postcondition is asserted, its effects are propagated immediately. However, sometimes this is not desired. In the search dialogue shown in Figure 1, for instance, the value set by means of the text item should take effect only if and when the ‘ok’ button is pressed. To make this possible widgets can propose predicates in their postconditions. This means, when the postcondition is asserted, the proposed predicates do not take effect immediately; they are merely recorded as being proposed. Another widget, usually an ‘ok’ button, can then accept the proposed predicates (as in Example 5).

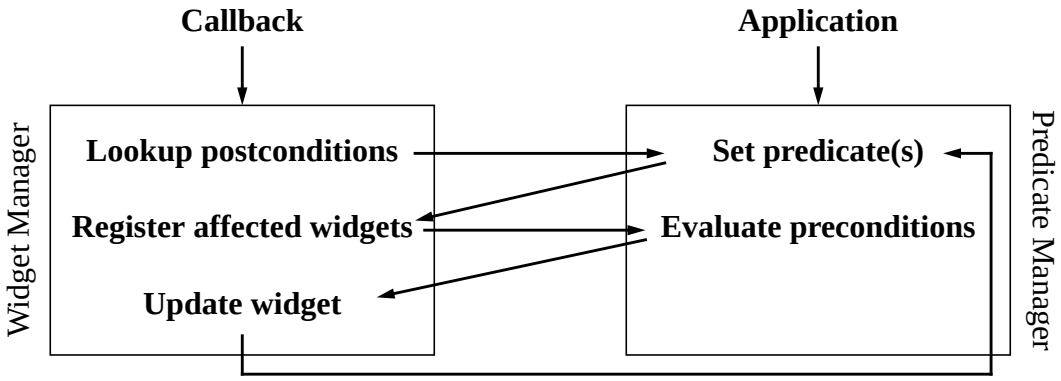

Figure 3 Interaction between widget manager and predicate manager

Numeric text item *search_track_item*
pre visible: *visible(*SEARCH_WINDOW*)*
pre enable: *current_track(set_value)*
post changed: *propose current_track(get_value)*

Button *search_ok*
pre visible: *visible(*SEARCH_WINDOW*)*
post select: *accept search_track_item:proposed*

Example 5 Proposed predicates - when the value of the text item is changed a new current track is proposed, but not yet set. When the ok button is selected the proposed change is asserted and the current track is changed.

Architecture

The pre- and postconditions are handled by two main components: the predicate manager and the widget manager (Figure 2). These two components form a transparent layer between the application[1] and the toolkit. It is transparent because the application does not need to know about the pre- and postconditions if it does not make use of them.
The predicate manager is in charge of the CSB; it is the only part of the system that can write to and read from the CSB. The application is allowed to make changes on the CSB only by calling functions in the predicate manager. This way the predicate manager can always make sure all widgets are updated when there is a change on the CSB. This also allows for changes (in future versions) in the organization of the CSB and the algorithm that evaluates predicate expressions (preconditions). The predicate manager also takes care of parsing and evaluating pre- and postconditions.
The widget manager registers widgets that have pre- and postconditions associated with them. Whenever a widget with pre- and postconditions is to be created, the widget manager asks the UI toolkit to create a normal widget and registers its id, name, pre- and postconditions and predicate variables and stores this information in a ‘shadow widget’. The pre- and postconditions supplied by the programmer as text strings are parsed by the widget manager before they are added to the ‘shadow widget’. It also adds the widget’s id to a hash table used to find widgets that might need to be updated as a result of changes in the CSB (this is described later in this section). The widget manager also handles communication with the UI toolkit. When the application wants to retrieve or change widget attribute values, the widget manager first checks if the attributes are of interest to the pre- and postcondition mechanism: In addition to pre- and postconditions, which can be changed and retrieved as normal attributes, the widget manager also needs to know about callback routines, because it has to intercept all callbacks. When a user interacts with a widget, the underlying UI toolkit generates a callback which invokes a function in the widget manager (Figure 3). The widget manager looks up the postconditions of the corresponding widget and asks the predicate manager to make changes to the CSB. After the predicate manager has made changes to the CSB, it asks the widget manager to supply a list of widgets that might be affected by the changes. For every widget in this list the predicate manager reevaluates its preconditions and informs the widget manager about changes in the state of that widget, after which the widget manager asks the UI toolkit to update the widget. After the state of a widget has changed, its children might need to be reevaluated, for a widget cannot be visible if its parent is not visible. Therefore these children are added to the list of potentially affected widgets. The process then continues until this list is empty.

While we have not done efficiency testing, we have not noticed any slowdown in the example interfaces we implemented. However, these interfaces were relatively small (less than 100 widgets). We are not concerned about efficiency, because the current implementation, which does not slow down small applications, is not the most efficient implementation and can be improved in several ways.

1. The system was originally developed as part of UIDE, where the UIMS part takes care of all the user interface related tasks. However, this system can also be used independently of UIDE. When we talk about the application we mean the application and/or UIMS.

Extending Developers Guide With Pre- And Postconditions

Sun Microsystems' Developers Guide (DevGuide) [SUN90] is a user interface layout tool. With DevGuide the designer can easily layout a user interface of an application by simply placing and dragging interface objects, such as windows, buttons, sliders, etc. Its major drawback is the inability to incorporate run-time dynamics, such as controlling the (dis)-appearing of popup windows. Interface Architect [HEWL90] has a built-in C interpreter, which allows the designer to write C callback functions. However, when the designer is laying out a user interface, he usually does not want to write code. In Interface Builder [NEXT90] and in version 3.0 of DevGuide it is possible to create 'connections' between interface objects. The designer can drag a connection, displayed as a rubber band line, between two widgets and specify the behavior of the connection. However, this does not address the important need to associate context-specific conditions with the connection, so that it will occur only when the conditions are true. For some dynamic behavior, semantic information about the state of the application is needed, to make the behavior depend on the current run-time context.

We are expanding DevGuide to allow the specification of pre- and postconditions. The attributes of widgets created with DevGuide are defined by property sheets, in which the designer can specify properties, such as label text, size, position, color, etc. These property sheets will be extended to include slots for pre- and postconditions. Once this is done, the designer will be able to specify and test run-time dynamics using DevGuide.

Another useful extension is the mapping of the connections onto pre- and postconditions: the designer specifies connections, while the system generates pre- and postconditions from these connections. This includes the connections in the pre- and postcondition mechanism and allows for better fine tuning; pre- and postconditions generated from connections can be edited, so extra conditions can be added to the connections.

To take full advantage of the pre- and postconditions, support for the designer to make changes to the CSB in test mode to simulate changes which would normally be made by the application itself, should be provided.

Conclusions & Future Work

The pre- and postcondition mechanism is useful for prototyping user interfaces, especially in conjunction with other user interface design tools. It allows a designer to change parts of a user interface without affecting other parts of the interface. It also allows testing of program dynamics in an early phase of a design. The pre- and postconditions can also be used at run-time to automatically generate help and explanations.

With the designed architecture presented in this paper, the mechanism can easily be integrated in different environments and can be made to work with a variety of tools and toolkits.

As we mentioned before, efficiency does not appear to be a critical issue. However, if better efficiency is needed, several improvements can be made:

- The preconditions are currently evaluated by a rather inefficient backtracking algorithm, which can be replaced by a more efficient algorithm.
- When changes are made to the CSB, the system generates a list of possibly affected widgets. A linear hash table is used to search for these widgets. Using a search tree would improve the time to find these widgets.
- The CSB itself is currently organized as a sorted linear list and determining the truth value of a predicate requires a linear search. The use of a tree structure for the CSB will improve the time to determine the truth value of a predicate.
- Widgets in an interface are organized in a tree structure; most widgets have a parent widget. The system currently uses this information to make sure a child of a non-visible widget is not made visible. This information can also be used to limit the number of widgets of which the preconditions should be reevaluated.

Currently, only one CSB is used for each application. Although this has some advantages, it means all predicates are global. For the same reasons why scoping is used in programming languages, we would like to have some form of scoping for predicates used in pre- and postconditions.

In the section on DevGuide, we mentioned extensions that are going to be implemented soon. Once the pre- and postcondition mechanism is integrated with DevGuide we would like to improve the user interface for specifying pre- and postconditions. This can be as simple as supplying lists of commonly-used predicates from which the designer can select the desired predicates, or as complex as a graphical dialogue editor in which a dialogue is represented by augmented transition networks or petri nets, which can be converted to pre- and postconditions.

Finally we plan to develop a graphical debugging tool, where the dialogue, described by pre- and postconditions, is presented in a graphical form and where the run-time behavior can be traced and possibly modified.

Acknowledgments

Daniel Gieskens would like to thank Jim Foley for creating an excellent research environment. We would like to thank the many George Washington University and Georgia Tech students who provided feedback and help in numerous ways: Dennis de Baar, Won Chul Kim, Piyawadee (Noi) Sukaviriya, Srdjan Kovacevic, Lucy Moran and Jens Kilian. We also would like to thank prof. Hikmet Senay at the George Washington University for his help with matters related to predicate logic.

Partial funding for this work was provided by Sun Microsystems' collaborative research program, and by National Science Foundation Grant # IRI-8813179. We thank Bob Ellis of Sun for his skillful management of our research proposal, and Bob Watson, development manager of Sun's DevGuide, for his support and interest.

References

FOLE87 Foley, J., C. Gibbs, and W. Kim, "Algorithms to Transform the Formal Specification of a User-Computer Interface" in Proceedings INTERACT '87, 2nd IFIP Conference on Human-Computer Interaction, Elsevier Science Publishers, Amsterdam, 1987, pp. 1001-1006.

FOLE89 Foley, J., W. Kim, S. Kovacevic, and K. Murray, "Defining Interfaces at a High Level of Abstraction", IEEE Software, 6(1), January 1989, pp. 25-32.

FOLE91a Foley, J., D. Gieskens, W Kim, S. Kovacevic, L. Moran, P. Sukaviriya, "A Second-Generation Knowledge Base for the User Interface Design Environment", Report GWU-IIST-91-13, Dept. of Electrical Engineering and Computer Science, George Washington University, Washington D.C., May 1991.

FOLE91b Foley, J., W.C. Kim, S. Kovacevic, and K.Murray, "UIDE - An Intelligent User Interface Design Environment", in Sullivan, J. and Tyler, S. (eds.), *Architectures for Intelligent Interfaces: Elements and Prototypes*, Addison-Wesley, 1991.

GREE85 Green, M., "The Design of Graphical User Interfaces", Technical Report CSRI-170, Computer Systems Research Institute, University of Toronto, 1985.

GREE86 Green M., "A Survey of Three Dialogue Models" in ACM Transactions on Graphics 5(3), July 1986, pp. 244-275.

HELL90 Heller D., "XView Programming Manual", O'Reilly & Associates, Inc., October 1990, ISBN 0-937175-52-8.

HEWL90 Hewlett-Packard Company, "HP Interface Architect Developer's Guide", Hewlett-Packard Company, Corvallis, Oregon, October 1990.

NEXT90 NeXT Computer, Inc., "NeXTstep Concepts", NeXT Computer, Inc., Redwood City, CA, 1990.

PITT90 Pittman J., and C. Kitrick, "VUIMS: A Visual User Interface Management System" in Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Snowbird, Utah, October 1990, pp. 36-46.

SENA89 Senay H., P. Sukaviriya, L. Moran, "Planning for Automatic Help Generation", Report GWU-IIST-89-10, Dept. of Electrical Engineering and Computer Science, George Washington University, Washington D.C., 1989.

SUKA90 Sukaviriya P., and J. Foley, "Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help" in Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Snowbird, Utah, October 1990, pp. 152-166.

SUN90 Sun Microsystems, Inc., "Open Windows Developer's Guide 1.1, Reference Manual", Part No. 800-5380-10, Revision A, of June 1990.